Work Stealing
in
Multiprogrammed Environments

Brice Dobry
Dept. of Computer & Information Sciences
University of Delaware
Outline

• Motivate the issue
• Describe work-stealing in general
• Explain the new algorithm and the problems along the way
• Demonstrate its effectiveness
Realistic Parallelization

Processes vs Speedup

- Ideal
- Realistic

CISC 879 : Software Support for Multicore Architectures
Limited number of processors available!

- Ideal
- Realistic

![Graph showing speedup vs. number of processes](image)
What Actually Happens

![Graph showing speedup with processes]

- **Ideal**
- **Realistic**

*Processes vs. Speedup*
• Bad scheduling of the processes causes poor utilization

• Worst Case (5 processes, 4 processors)

<table>
<thead>
<tr>
<th>Processes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
</tbody>
</table>
What Actually Happens

- Four processes run simultaneously to completion
- Fifth process waits for resources

```
P0  T0
P1  T1
P2  T2
P3  T3
```

Time

CISC 879 : Software Support for Multicore Architectures
What Actually Happens

- Fifth process runs once one of the other processes completes
- 3 processors have only 50% utilization
Some Actual Results

- Testbed has 8 processors
- Work is statically partitioned
Other Solutions

• Co-scheduling
  • All processes of a program are scheduled to execute at the same time
  • Gives the appearance of a dedicated machine
Other Solutions

• Process Control
  • Kernel notifies program of current resources
  • Processes are dynamically created or killed based on what is available
  • \# processes = \# processors
Better Solution

- Work Stealing
  - Dynamically balance the load across processes
  - Maintain utilization even when competing with other programs for resources
  - Handle even worst possible scheduling of processes
**Term Definitions**

- **Process**
  - A kernel-scheduled entity; all processes of one program can share memory

- **Thread**
  - User-level task scheduled by the user-level library
Work-Stealing Algorithm

- Each process maintains a thread deque
  - All threads in this deque are ready to run (not blocking on anything)
- Running threads can unblock other threads, or create new threads
  - New (or newly unblocked) threads are added to the bottom of the deque
Work-Stealing Algorithm

- When a thread completes execution, the next thread is pulled off the bottom of the deque.
- When a process’s deque is empty, it will steal work from another process.
  - Threads are “stolen” from the top of the other process’s deque.
Work-Stealing Algorithm

0 1 2
3 4 5
Work-Stealing Algorithm

CISC 879 : Software Support for Multicore Architectures
Work-Stealing Algorithm

0

1

2

3

4

5
Work-Stealing Algorithm

CISC 879 : Software Support for Multicore Architectures
Work-Stealing Algorithm
Work-Stealing Algorithm
Work-Stealing Algorithm

CISC 879 : Software Support for Multicore Architectures
Implementation

- Difficult issues
  - Synchronization between processes
  - Prevent bad kernel scheduling from deteriorating performance
1. Protect the deques with spinlocks
1. Protect the deques with spinlocks

- One process could be preempted by the kernel while it has a lock
- A second process attempting to get that lock will spin on it until the first process resumes and frees the lock
1. Protect the deques with spinlocks
2. Protect deques with blocking locks
2. Protect deques with blocking locks

- One process gets preempted while it has the lock
- Second process attempts to get the lock, blocks, yielding the processor
- First process resumes execution and releases the lock then eventually is preempted
- Second process can now obtain the lock
2. Protect deques with blocking locks
3. Use atomic operations to operate on the deques and avoid locks altogether
3. Use atomic operations to operate on the deques and avoid locks altogether

- Running processes empty all deques
- Only runnable thread is on a process that has been preempted
- Running processes continuously make failed stealing attempts
- Eventually, the preempted process runs and threads are unblocked
3. Use atomic operations to operate on the deques and avoid locks altogether
Implementation - Scheduling

- Prevent processes from wasting resources spinning on failed steal attempts
  - Use the system calls to help the kernel schedule processes more conveniently
    - priocntl
      - Change the priority level of this process
    - yield
      - Voluntarily release the processor
Deque empty?  

Yes → Lower scheduling priority  

No → Pop thread and execute  

Raise scheduling priority  

Success → Attempt to steal  

Fail → Yield
Final Results

The graph shows the speedup of different programs as the number of processes increases. The programs include ideal, mm(1024), lu(2048), barnes(16K,10), heat(4K,512,100), msort(32M), and ray(). The x-axis represents the number of processes, while the y-axis represents the speedup. The ideal speedup line is shown in gray, and the other lines represent different programs, each with a unique pattern and color.
Conclusions

- This work-stealing algorithm proves to be effective and efficient
  - Performs as well as static partitioning in a dedicated environment
  - Far outperforms static solutions in non-dedicated multi-programming environments
  - User-level implementation with no kernel support required
Questions?